On efficient black box optimization of systems defined by 100 or more parameters

Olivier Goury, Sven Ahlinder

On efficient black box optimization of systems defined by 100 or more parameters

Olivier Goury, *Chalmers University of Technology, goury@student.chalmers.se*
Sven Ahlinder, *Volvo Technology Corporation, sven.ahlinder@volvo.com*

April 29, 2010

Abstract

In product development, the optimization of hectovariate systems (that is, depending on more than a hundred variables) is hard to handle with usual black box methods, since they require a huge number of experiments. That is an issue since each experiment can be both expensive and time-consuming. Furthermore, these methods can't even handle such big problems within a reasonable period of time. The Lean optimization algorithm [1] was one approach to overcome these difficulties. In this paper, we will come up with a new lean algorithm, inspired from [1] and developing new ideas. It will then be tested on some classical mathematical problems of various sizes and a practical example of combustion simulation engine issued from Volvo Technology. We will see that this new algorithm has a faster and more robust convergence, and is able to significantly improve most systems within a number of experiments less than the number of variables.

Keywords: Lean optimisation, Supersaturated design, Hectovariate problems, Engine

1 Introduction

Black box systems are such that their constituents are unknown and one can't apply any heuristics to optimize them. The only way to get information is to perform experiments. When these are expensive and time-consuming (like in many complex industrial models), they should be very carefully planned.

In **DoE** (Design of Experiments) **optimization**, an experimental plan is made with the idea of getting as much information as possible. From that information, important parameters can be found and the system can be optimized in some way. The main problem is that it requires a number of experiments increasing exponentially with the number of parameters, and is hence not applicable if the system we study is hectovariate or larger.
Efficient global optimization(**EGO**), described in [7], is trying to create a good model of the system by perfoming the experiments sequentially, in a way that new experiments are smartly performed where the model is likely to be the less accurate (using a statistical analysis), until reaching a predefined approximation error. That requires about 10 times more experiments than the dimension

of the system. Again, that algorithm starts to be unable to handle problems when the number of parameters gets above 30 (the running time becomes unreasonable and the solution gets worse when you increase the size of the problem).

Another technique is to use a **gradient algorithm**. By making small changes for each variable but one at a time, one can get a good approximation of the gradient at the current point. One can then take a step following that direction and re-evaluate the gradient. For each step, this method requires as many experiments as the dimension of the sytem.
One can also be tempted to use **Newton's method**, which is known to be more robust than the steepest descent algorithm for example. However, to get an approximation of the hessian, a number of experiments proportional to the square of the number of parameters is needed for each step.

All those methods require a lot of experiments (In this article, we consider that the number of experiments is "a lot" when it is larger than a small multiple of the number of variables in the system). In lean optimization, we try to do as a whole very few experiments (even less than the number of degrees of freedom of the system when that is possible) and still find a satisfying optimum. The algorithm performs experiments sequentially, using a special branch of DoE, which are supersaturated designs (designs where the number of experiments is less than the number of variables).

We present more precisely which problems we are interested in in the next section. In section 3, we explain the basic concept behind lean optimization. An analysis of supersaturated design is made in section 4. Section 5 shows ways to improve the lean algorithm from [1] and presents the new algorithm. Section 6 will present the performance of the algorithm on both theoretical and practical examples, before section 7 where conclusions will be drawn.

2 Type of problem considered

The type of problem we are interested in must contain at least 100 degrees of freedom. We assume that each degree of freedom has a known range. The goal is to optimize that system as much as possible with a very limited number of experiments. The type of optimization problem we want to solve can be formulated this way:

$$\min f(X) = f(x_1, x_2, ..., x_N)$$

$$L^{low} \leq X \leq L^{up} \qquad (\Leftrightarrow l_i^{low} \leq x_i \leq l_i^{up}, i = 1, ..., N) \qquad N > 100$$

$$c(X) \leq \mathbf{0}$$

where $c(X) = \begin{pmatrix} c_1(X) \\ c_2(X) \\ \vdots \\ c_p(X) \end{pmatrix}$ is the constraints function that can be linear or non-linear.

Two extra specifications make the problem more difficult:

- Functions f and c are black boxes: the only way to get information about them is to perform experiments.
- Performing one experiment to evaluate f or the constraints c is expensive. Hence, experiments must be performed parsimoniously. More precisely, we assume the number of affordable experiments is about N (the number of variables).

For most cases, it is not possible to find the actual optimum within so few experiments. The goal is then to find a set $X = x_1, x_2, ..., x_N$ which optimizes f much more than would the best out of N random tries do (randomly picking N sets X_i within the predefined range, evaluating $f(X_i)$ and picking the best one).

3 What is the idea behind lean optimization, and why have we hope for it to work?

In the lean optimization algorithm, we perform very few experiments at a time , and we fit the results of these experiments to a linear model, that will then give us a direction to follow. Since we have to do several steps, and that we try to do less experiments that the number of variables as a whole, we have to use designs extremely supersaturated for each step (less than the square root of the number of variables).

For example, if the function f you want to optimize has N variables $x_1, x_2, ..., x_N$, you seek $N + 1$ coefficients $\beta_0, \beta_1, ..., \beta_N$ so that

$$f(x_1, x_2, ..., x_N) \approx \beta_0 + \beta_1 x_1 + ... + \beta_N x_N \text{ for } (x_1, x_2, ..., x_N) \in \text{current area.}$$

If one performs m experiments for each step, one gets a linear system of m equations and $N+1$ variables to solve (after some scaling on the x's):

$$\begin{pmatrix} 1 & -1 & 1 & 1 & -1 & \dots & -1 \\ 1 & 1 & -1 & 1 & -1 & \dots & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \dots & -1 \\ 1 & 1 & -1 & 1 & 1 & \dots & 1 \\ 1 & -1 & -1 & -1 & -1 & \dots & 1 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ \vdots \\ \beta_{N-1} \\ \beta_N \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{m-1} \\ f_m \end{pmatrix}$$

which can be simply written in the form

$$\begin{bmatrix} \mathbf{1} & D \end{bmatrix} \beta = \tilde{D}\beta = F$$

where D is the design matrix (The first column is always made of 1's since that corresponds to the coefficient β_0. A scaling is made for each step so that D is always made of -1's and 1's).
This system is very highly underdetermined since we choose m much smaller than N. This means not only that there are infinitely many solutions but also that the solution space is of rather high dimension compared to the dimension of the problem. We solve it by using the Moore-Penrose pseudoinverse [8] and [9]: $\beta = \tilde{D}^+ F$. This gives the solution with minimal norm. This implies all variables are treated with the same importance (Since they are all scaled between -1 and 1), and that seems a fair choice, since we have no a priori knowledge about the influence of each variable. Despite that, one can legitimally get worried on how bad the gradient approximation will be, since you have to pick a solution vector β out of a very large set. One could think that the accuracy of the solution β is on average proportional to $\frac{\text{number of experiment}}{\text{number of variables}}$. Ahlinder and Gustafsson [2] showed that the average accuracy one gets is actually $\sqrt{\frac{\text{number of experiments}}{\text{number of variables}}}$ for a well choosen design (Here, the accuracy measure is the correlation between what is the real direction and what direction is actually calculated, that is, the cosine of the angle between the two directions).

For example, consider a problem depending on 200 variables, and try to find its optimum with the lean algorithm by performing only 10 experiments per step. For each step, it could be conjectured that the direction obtained from the regression analysis is $\frac{10}{200} = 5\%$ in the right direction (this means the correlation between the right direction and the one calculated is 0.05, that is, the cosine between the two directions is 0.05) but from [2] we get that it is about $\sqrt{\frac{10}{200}} \approx 22\%$. So we can see that despite a very large saturation, the direction taken is not as bad as expected. Instead of doing 1 step 100% in the right direction (which requires 200 experiments), we will do several steps 22% in the right direction (that requires only 10 experiments times the number of steps). One can hope that few of these cheap steps are sufficent to reach the same result. All the lean strategy of the algorithm is based on that concept.

4 Supersaturated designs: which are the good ones?

4.1 Why supersaturated design?

In DoE, a design is a matrix representing a number of experiments to be performed. A column is assigned to each variable and each row represents one set of values for each variable, that we call an **experiment**. In a 2-level case, we evaluate each variable only at its lowest and its highest value, represented respectively by -1 and $+1$ in the design matrix.

In a 2-level case, one tries to find a linear fit of the system studied. In usual cases, where experiments are cheap, one performs many experiments and then find the linear approximation that is the most consistent with the experiments for example in a least square sense. In a **saturated** case, the number of experiments performed is such that there is exactly one linear solution that fits all the experiments. That happens when the number of rows in the design is equal to the number of columns plus one. In a **supersaturated** case, the number of rows is smaller than the number of columns, and to find the linear approximation, one must actually solve an underdetermined system, which implies that there are more than one solution. Supersaturated designs are such that

the number of experiments performed is not large enough to caracterize the system by its first order approximation. These kind of designs are used when one can only afford to make a couple of experiments but still wants get a partial knowledge of the system's behaviour. Several strategies to derive 2-level supersaturated designs have been proposed in the last decades [3] [4] [6]. Since 2000, those have much less studied and the interest went into multi-level supersaturated design and especially mixed-level ones (See for example [5]). We are more interested in 2-level supersaturated designs in this paper and we will present a new method to construct them in the following section.

4.2 How to find a good supersaturated design?

The choice of the supersaturated design used in the lean optimization algorithm has a dramatic influence on the performance. Indeed, it dictates how experiments are performed, and a bad supersaturated design would provide too little information to choose effective directions and would lead to the failure of the algorithm. Which supersaturated design will give the best result?

In the 2-level saturated case, it is known that an optimal design should have orthogonal columns (Here, on a design matrix D, each column corresponds to one variable, and each row to one experiment). In the supersaturated case, since there are more columns than rows, it is impossible to have all the columns orthogonal. Hence, it is a common idea to say that they should be nearly-orthogonal instead, which means that there is a low pairwise correlation between them. This criteria is called $E(s^2)$([4]).

Ahlinder and Gustafsson [2] showed that this criteria is actually not relevant, but that the row rank is, since the solution of $D\beta = F$ found by the pseudoinverse is a projection of the real solution on the row space of D. That implies that the larger the row space of D is, the closer to the real solution you can be. In [1], Siomina and Ahlinder used a design generated by Lin's algorithm ([4]). [2] tells us that we have no guarantee on its quality, since it seeks a design with orthogonal columns instead of orthogonal rows. Furthermore, it gets less efficient if the design is very much supersaturated.

However, one can easily build much more efficient designs in this way (the idea comes from Ahlinder and Gustafsson):

create first a full factorial design, then transpose it and take the first half of the columns away (this is done to avoid an almost total correlation between the columns: the two halves are indeed identical except for the first row). Then, among the remaining columns, keep the ones that contains a ratio of 1's and -1's equal to 1 (or almost 1 if the columns are of odd length). We will refer to it later by the name of **AG design**, as for Ahlinder and Gustafsson.

For example, take a full design of order 4:

$$\begin{pmatrix} -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 \\ -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Transpose it and keep only the 2^{nd} half of the columns:

$$\begin{pmatrix} -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Then keep the ones with as many 1's as -1's:

$$\begin{pmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Of course that design is not supersaturated (with 4 rows for only 3 variables), but the saturation increases fast with the number of rows. For 10 rows, the design has 126 columns.

That design is actually the most supersaturated we found that is still giving enough information for a relevant regression analysis to be done. Indeed, the most important property of a design is that each column should have a ratio of -1's and 1's very close to 1. For a fixed number of rows like m, the best one can do is to simply take all possible columns with $\frac{m}{2}$ 1's and $\frac{m}{2}$ -1's (or for example $\frac{m-1}{2}$ 1's and $\frac{m+1}{2}$ -1's if N is odd). Given that set of columns, one can then improve the information gain by adding one more row of 1's, since that row will have the nice property of being orthogonal to all the others. The procedure described above is actually just giving the same result, with some nice ordering on the columns (if you consider that each column is a binary number by changing -1 with 0) which will be used in a following section.
Here is another example, with a 6 rows design. First take all possible columns of 5 elements with say two 1's and three -1's and classify them in order (a decreasing binary order if you exchange -1 with 0):

$$\begin{pmatrix} 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 \\ -1 & -1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 \end{pmatrix}$$

And then add one row of 1's:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 \\ -1 & -1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

That design is actually identical to the **AG design**.

Some properties of the AG design

- A design with $2p$ rows (respectively $2p+1$) has $\begin{pmatrix} 2p \\ p \end{pmatrix}$ columns (respectively $\begin{pmatrix} 2p+1 \\ p \end{pmatrix}$). The saturation increases rather fast with p.

- Each column has the same number of -1's and 1's in an even case (or p (-1)'s and $(p+1)$ 1's for an odd case). Experience shows that this property is necessary for a stable regression analysis. It could be explained by the fact that each variable has to be evaluated fairly from its smallest values to its highest values.

- Its rank is not full. It is the number of rows minus one. That is the price to pay for having a fair ratio of -1's and 1's for each column. This property indeed implies that the rows of the design are not linearly independant (since their sum is zero).

- Its columns are rather strongly correlated. It might be thought as a drawback, according to [4], but it is not at all, as will be seen in section 6. This will strengthen the idea developed in [2].

- It is column-ordered: the column are classified in order if you consider each column to be a binary number (exchanging all -1's with 0's). This property will be crucial for a strategy explained in section 5.2.2.

For a fixed number of variables N, one can get its design by choosing the number of rows that gives a number of columns the closest from above to N, and then delete as many columns as needed. For the sake of keeping the rows as different as possible, it is better to perform the deletion of these extra columns all other the design and not simply on the last columns. For example, if the problem has 202 parameters, you can choose the AG design with 11 rows (since the one with 10 rows only has 126 columns). It has 252 columns, so you have to delete 50 columns to fit the design to the problem. You could just delete the last 50 columns, but we expect the design to be giving more information on average if you delete a set of columns which is spread out all other the design matrix, for example the 5^{th} column, the 10^{th} column, ... the 250^{th} column.

5 Lean optimization algorithm: extension to constrained problems and improvements

5.1 Extension to constrained problems

In [1], the algorithm was presented in the unconstrained case. We extend it to the constrained case in this paper. What the algorithm of [1] is doing in each step is first selecting a sub-area of experimentation inside the domain, then performing some experiments on that sub-area according to the design chosen. Through regression analysis, the system is fitted to a local linear approximation, and from that analysis, another area of experimentation is chosen.
In the constrained version of the algorithm, a linear approximation of the constraints by regression is made as well, and the new area of experimentation is chosen so that it is around a point that optimizes the approximation of the function under the approximation of the constraints.

Of course, since the constraints are approximated, we can't guarantee that the local solution found will not be unfeasible, but we can hope for it to be at least close to feasibility, and not misleading the algorithm.

We end up with an LP problem (both the objective and the constraints are linear). It becomes then possible to apply traditionnal LP-solvers, such as the simplex algorithm or interior point methods. The function *linprog* from Matlab is used.

Here is a simple 2D example to illustrate the algorithm. It does not make much sense to apply the algorithm in that case, since it is not a supersaturated case, but it still gives a more concrete idea of what is happening.
Consider the minimisation problem

$$\min_{(x_1,x_2)} f(x_1,x_2) = (x_1 - 0.7)^2 + (x_2 - 0.7)^2$$

$$-\frac{9}{2}x_1^2 - x_1 + 3x_2 + \frac{1}{2} \leq 0$$
$$-1 \leq x_1 \leq 1$$
$$-1 \leq x_2 \leq 1$$

For the starting experiment area, we select $\frac{1}{3}$ of the range of each parameter, as represented in figure 1. You can also see the unfeasible zone indicated by the constraint in figure 2.

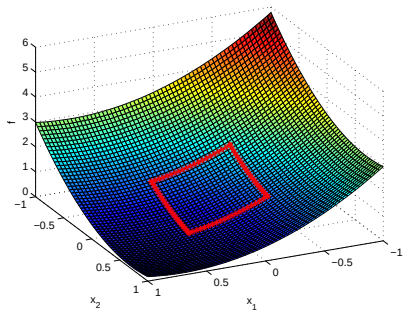


Figure 1: Objective function and experimentation area for step 1

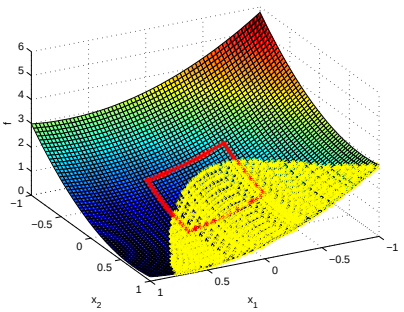


Figure 2: Objective and unfeasible area for step 1

After performing some experiments according to a certain design(for example the corners), we create a local linear approximation of the objective and of the constraint. We then find the optimum of this linear optimization problem on the current area and select the point found as the center of the new experiment area for step 2. So in our example, we get the minimization problem:

$$\min_{(x_1,x_2)} \hat{f}(x_1,x_2) = -1.4x_1 - 1.4x_2$$

$$-x_1 + 3x_2 \leq 0$$
$$-\frac{1}{3} \leq x_1 \leq \frac{1}{3}$$

$$-\frac{1}{3} \leq x_2 \leq \frac{1}{3}$$

The solution is $x_1 = \frac{1}{3}$, $x_2 = \frac{1}{9}$ (Please note that you don't need to perform any new experiments on the original objective function to find that optimum). Those linear approximations, as well as the optimal point, are represented in figure 3. In figure 4, you can see where that point actually is in the real problem.

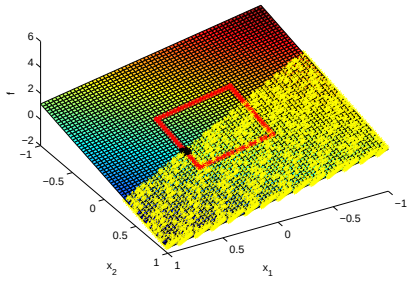


Figure 3: Linear problem and optimal point

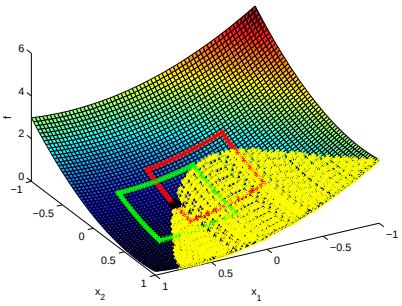


Figure 4: Real problem and experiment area selected for next step

In that case, it happened that the point choosen as the center for the new area is actually the best you could choose, since the linear approximation of the functions is accurate enough for the solution of the linearized problem to be the same than the original one. This almost never happen when you consider a supersaturated problem with much more dimensions: in those cases, you just expect to go in some percentage of the right direction (This percentage depends on how supersaturated you chose your design).
Then, everything starts over again with the new experiment area. Note that it is important to decide on a "shrinking strategy", that is to decide when and how much you should shrink the area after each step.

We expect each new area to be "half-feasible", that is about a half of it is feasible, the other half is not (according to the linear approximations).

5.2 Improvements

5.2.1 Shrinking strategy

In [1], the shrinking strategy is systematic. The experimentation area size in [1] is multiplied by one coefficient at each step (for example $\frac{2}{3}$), so that the total number of steps performed by the algorithm is typically less than 10, since beyond that, the area gets very small and the moves are not noticeable anymore.
In this paper, we choose the first experimentation area to be rather large (for example $\frac{1}{5}$ of the whole size of the domain). Here, large means that few steps will be necessary to reach a region around the optimum. We keep its size constant until the direction of the last 2 moves are judged to be in a "nearly" opposite direction (for example when their correlation is smaller than $-\frac{1}{2}$). The

idea is that when that happens, you have reached some place close to the optimum, but you are going back and forth since your area is too large. Then it becomes interesting to shrink it. This is done as often as needed. You could start with a small area at first, but the convergence would be slower, which means extra experiments would be performed.
See figures 5 and 6, where we optimize some simple quadratic function in 2D, in the unconstrained case. We take the origin as initial point (the center of the first experimentation area). The star represents the optimum.

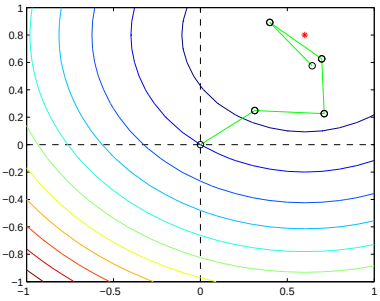


Figure 5: The last 2 steps were sensibly in opposite direction

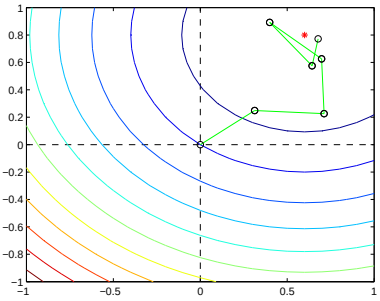


Figure 6: Then we take a smaller length for the next step

5.2.2 Design permutation

A simple way of improving the performance of the algorithm is to randomly permutate the columns of the design chosen after each step. That is a way to perform every time experiments that are very different, and broaden the directions we take. Experience shows that not permuting the columns leads to optimize along a very narrow set of directions, which makes the optimization stop early in most cases. This is what was done in the previous version of the lean algorithm [1]: no permutation were used.

For example, let's assume we picked the design matrix $D = [d_1 d_2 ... d_N]$ to dictate our experiments, where d_i is the i^{th} column of D. In the original version of the algorithm [1], D was used at each step to perform the experiments. In the random permutation version, it is not D which is used, but a shuffled copy of it: $\hat{D} = [d_{\sigma(1)} d_{\sigma(2)} ... d_{\sigma(N)}]$. $\hat{D}$ is recreated at each step. To understand why this is likely to give better performance, we present here a simple problem (which be reanalysed in the next section 6):

$$\begin{aligned} \min_x f(x) &= \frac{1}{1000} \sum_{i=1}^{N} (x_i - i)^2 \\ -100 \leq x_i &\leq 250, \quad i = 1, ..., N \\ N &= 250 \end{aligned}$$

This function is simple sum of squares and its minimum is obviously 0, reached in

$x = \begin{bmatrix} 1 & 2 & \dots & 249 & 250 \end{bmatrix}$ In Figure 7, is displayed the convergence of the algorithm using the 2 strategies.

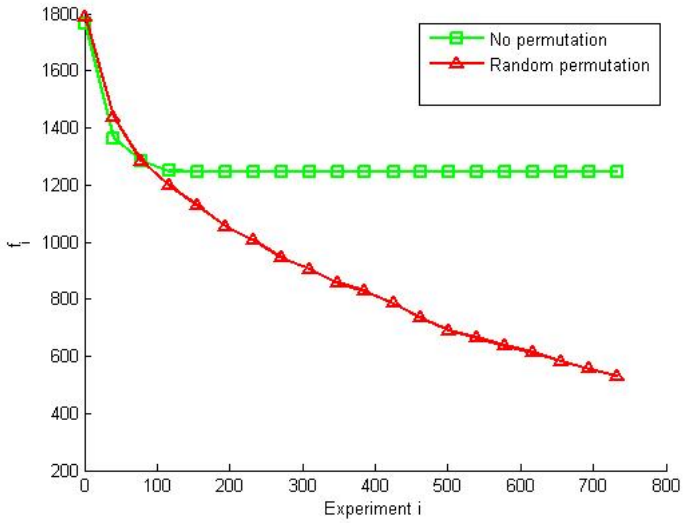
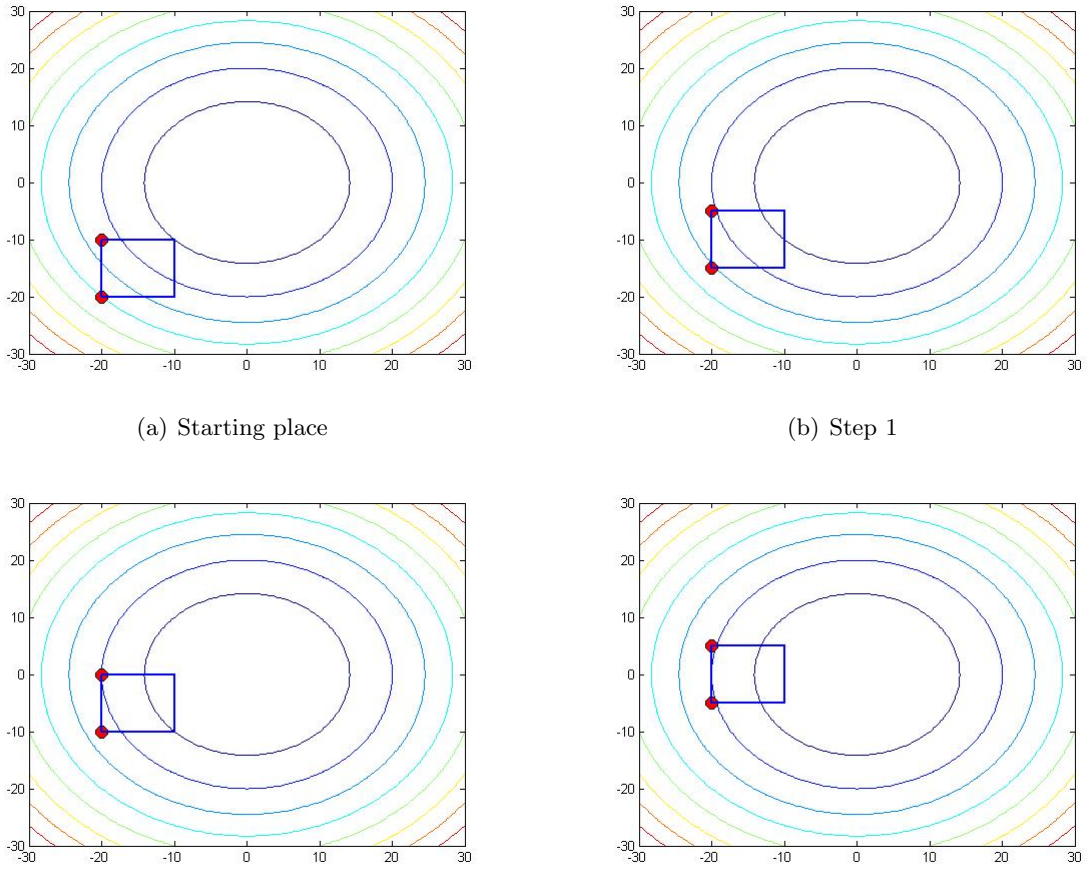


Figure 7: Convergence of the algorithm using the 2 strategies (for clarity, only the average of the experiments is printed)

We can see that when not performing any permutation on the design, the algorithm seems to reach some optimum far away from the real solution and not to move out of it. This does not happen with the random permutation. This phenomenon can actually be explained. To get some insights, consider the same problem but with $N = 2$ only (becomes a 2 dimensional problem). In Figure 8, we represent successives steps of the algorithm, assuming we picked a supersaturated design with 2 rows $D = \begin{pmatrix} -1 & 1 \\ -1 & -1 \end{pmatrix}$, leading to make 2 experiments in the left corners of the experiment area (represented in red in the figures).



(a) Starting place

(b) Step 1

(c) Step 2

(d) Step 3

Figure 8: 3 steps of the algorithm with a fixed design

Keeping exactly the same design leads to minimising along the y axis only, because it is always the 2 left corners which are experimented. The algorithm with random permutation would time to time perform some experiments on the left and right down corner (when the columns are permuted we get $D = \begin{pmatrix} 1 & -1 \\ -1 & -1 \end{pmatrix}$, which would make it possible for the experiment area to move along the x axis as well and hence find a better solution.

When working with a design such as the AG design (presented in section 4), another idea is to make a permutation which is not random, but following the order of the components of the gradient from the last step. Since the columns are in a particular order in that design, the idea is to "feed" it with variables that have some similar ordonancy. We will call that technique the **adaptive permutation**.

To get a bit more of understanding, assume β is the first order Taylor expansion of the function f on the current experiment area, and $\hat{\beta}$ the approximation of β we are trying to find on that same area. Then, we have $\begin{bmatrix} \mathbf{1} & D \end{bmatrix} \beta = \tilde{D}\beta = F$ and we seek $\hat{\beta}$ such that $\tilde{D}\hat{\beta} = F$ too. We use the

solution given by the pseudoinverse, that is

$$\hat{\beta} = \tilde{D}^{+}F = \tilde{D}^{+}\tilde{D}\beta$$

Since $\tilde{D}$ has more columns than rows, $\tilde{D}^{+}\tilde{D}$ is not the identity matrix, and hence $\hat{\beta} \neq \beta$ in general. We just know that the correlation between $\hat{\beta}$ and β is on average $\sqrt{\frac{\text{number of experiments}}{\text{number of variables}}}$ from Ahlinder Gustafsson [2]. Actually, experience shows that if D is column-ordered such as the AG design, and if the components of β are in order (for example $\beta_1 < \beta_2 < \ldots < \beta_N$; we don't care of β_0 corresponding to the constant part), then that correlation becomes much higher. The idea of **adaptive design permutation** is hence to reorder the design from the knowledge of the previous step (some $\hat{\beta}$ hopefully close to β) so that β gets a bit ordered with respect to this reordered design, implying that the $\hat{\beta}$ of the new step is closer to the β of the new experiment area.

More precisely, from the last β calculated, the permutation μ of $\{1, ..., 250\}$ is choosen such that $\beta_{\mu(1)} < \beta_{\mu(2)} < \ldots < \beta_{\mu(N)}$. Then the permutation μ^{-1} is applied to permute the columns of the original design, so that its column-ordering is following the one of the last β calculated.

For example, consider a simple problem with 3 variables and let's assume we chose the design $D = \begin{pmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$ from section 4, and that the gradient we got at the last step was $g = \begin{pmatrix} 5 \\ -3 \\ 1 \end{pmatrix}$ (We actually have $\beta = \begin{bmatrix} \beta_0 \\ g \end{bmatrix}$ where β_0 is the estimated average of the objective on the current experiment area). Then, we will apply to D the permutation that makes match its first column with the smallest value of g, the second column with the second smallest and so on. We hence get $D' = \begin{pmatrix} -1 & 1 & -1 \\ -1 & -1 & 1 \\ 1 & -1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$. Now, from the original design point of view, it looks like the gradient is $\begin{pmatrix} -3 \\ 1 \\ 5 \end{pmatrix}$, and the variables are classified in increasing variation order. This manipulation is done at the beginning of each step, always starting from the original design. In Figure 9, are compared the 3 permutations strategies on the test problem.

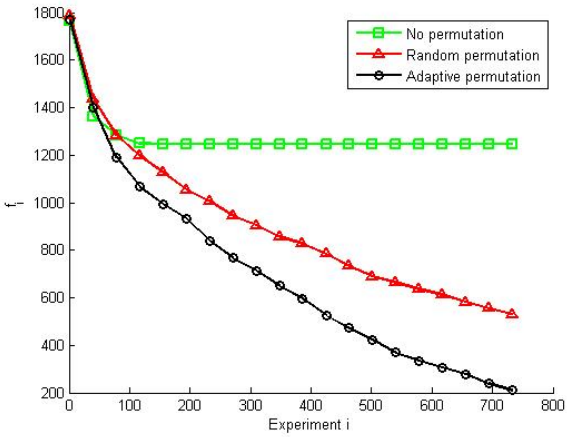


Figure 9: Convergence of the algorithm using the 3 strategies

We see that adaptive permutation improves the convergence even more. It is though hard to give a convincing reason for its good performance. Experimentation shows that it works well in most cases (check section 6.1.2).

5.2.3 Design size used

One issue of the lean algorithm is to choose a size for the design used for the experiments. We will here make some investigations that, even if they are not claiming to be a rigorous demonstration, will give a hint.

Unlike usual algorithms, whose trajectory can be ideally thought as a straight line going from the starting point to the optimum, the lean algorithm has a trajectory that can be thought as a kind of a **logarithmic spiral**. Indeed, each step is on average only right at some fixed low percentage, and we decrease the step size along its execution. See Figure 10 and 11.

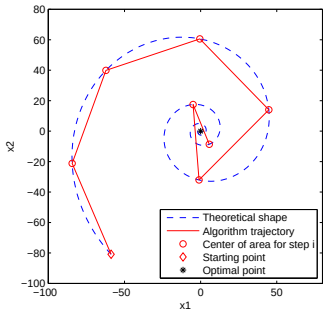


Figure 10: Representation of the behaviour of the algorithm with adaptive permutation in a parabolic 2D case

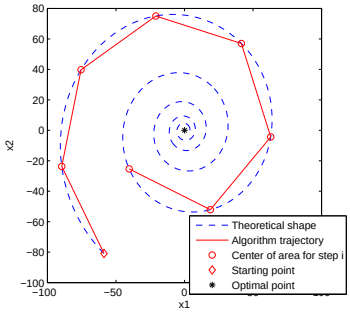


Figure 11: Representation of the algorithm trajectory for the same case, but with a smaller design (fewer rows)

In the limit case where the saturation is 1 (which means the number of columns equal to the number of rows - 1), the spiral just becomes a straight line from the starting point to the optimum.

The higher the saturation is, the more the spiral takes time to reach its center and so the more you need to do steps. On the other hand, having a high saturation means performing less experiments per step. Which size should the design have, between the leanest one (AG design) and a saturated one (with the number of columns equal to the number of rows - 1)?

At a fixed distance r from the center, the length of a spiral in 2D is finite and equal to $\frac{r}{cos(\phi)}$, where ϕ is the angle caracterizing how fast the spiral converge to its center (At any point of the spiral, it is the angle between the tangent and the radial line). Say now that the problem we are interested in has N variables. If you choose a design with m rows, then from [2], we get that $cos(\phi) = \sqrt{(\frac{m}{N})}$. So for a small step length l, the number of steps that has to be done to reach the optimum is roughly $\frac{1}{l}\frac{r}{\sqrt{\frac{m}{N}}}$. That implies that the number of experiments done is

$$(\frac{1}{l}\frac{r}{\sqrt{\frac{m}{N}}})m = (\frac{r}{l}\sqrt{N})\sqrt{\mathbf{m}}$$

We see that this number increases with m, the number of rows of the design. It is thus an advantage to make m as small as possible. **Choosing the AG design seems hence to be the best choice**, assuming that this stays true in higher dimensions.

5.3 New lean algorithm

In this part, we present what is intuitively the most promising version of the new lean algorithm. Consider a minimization problem as defined in section 2.

1. Pick the smallest AG design that fits the problem as the original design: D_{original}

2. Fix the current design to be the original design ($D = D_{\text{original}}$) and fix a starting experimentation area A to be centered in the domain and choose its size (for example 1/10 of the whole domain).

3. Perform experiments on the current area A according to the current design D

4. Evaluate β and β_{const}, respectively the first order Taylor approximation of f and c on A by solving $\begin{bmatrix} \mathbf{1} & D \end{bmatrix} \beta = \tilde{D}\beta = F$ and $\tilde{D}\beta_{const} = C$ (where F is the result of the experiments for f and C for the constraints c) using pseudoinverse: $\beta = \tilde{D}^{+}F$ and $\beta_{const} = \tilde{D}^{+}C$

5. Set D to be D_{original} but with a column-permutation following the order of β (That is **adaptive permutation**). That D will be used for the next step.

6. Find X_{sol}, the solution of the minimization problem, an LP-problem:

$$\begin{aligned} &\min_{X} \beta X \\ &\beta_{const} X \leq 0 \\ &X \in A \end{aligned}$$

7. Select the new experimentation area A to be centered in X_{sol} and shrink it if this last step is nearly opposite to the previous one.(**Shrinking strategy**)

8. Go back to step 3

For an unconstrained problem, the minimization problem in step 6 becomes trivial.

6 Numerical results

6.1 On classical mathematical functions

We first present the test examples and compare the performance of the new lean solver with a simple gradient solver. Then, we check our intuition about which design permutation strategy and which design size should be choosen.

6.1.1 Test problems

We here present several test problems and compare the result of a common gradient solver and the lean solver using **adaptive permutation** and the **AG design with as few rows as possible** (intuitively the most efficient strategy). Then, in section 6.1.2, we will check our intuition about which design size and which design permutation should be choosen on those same test examples.

Test problem 1: a parabolic problem

$$\begin{aligned} \min_x f(x) &= \frac{1}{1000}\sum_{i=1}^{N}(x_i - i)^2 \\ -100 &\le x_i \le 250, \quad i = 1, ..., N \\ N &= 250 \end{aligned}$$

This function is simple sum of squares and its minimum is obviously reached in $x = \begin{bmatrix} 1 & 2 & \dots & 249 & 250 \end{bmatrix}$ and it is zero. To make it more realistic (since the variables of a problem are not classified in the order of their optimal value in practice), we mess that problem up by introducing some permutation σ of {1,...,250}, so that the problem becomes:

$$\begin{aligned} \min_x f(x) &= \frac{1}{1000}\sum_{i=1}^{N}(x_{\sigma(i)} - i)^2 \\ -100 &\le x_i \le 250, \quad i = 1, ..., N \\ N &= 250 \end{aligned} \tag{1}$$

Now the optimal solution is $x = \begin{bmatrix} \sigma^{-1}(1) & \sigma^{-1}(2) & \dots & \sigma^{-1}(249) & \sigma^{-1}(250) \end{bmatrix}$. First of all, we have to choose a design. We take the AG design with 11 rows. It hence has $\binom{11}{5} = 252$ columns, and we delete 2 columns to fit it into the problem. Please note that it is the thinest design we could have used, since the one of size 10 only has 126 columns, which is not sufficient in this case.

Make sure you don't confuse the ***permutation*** σ *used just as a trick to mess up the problem and the* ***design permutation strategies*** *(No-permutation, Random permutation, Adaptive permutation) used to improve the convergence of the algorithm.*
In Figure 12 are displayed the comparative performance of the lean algorithm using adaptive permutation and the steepest descent algorithm. Here the steepest descent algorithm uses a gradient approximated by making very small variation for each variable one at a time (requires as many evaluations as variables) and then performing a linesearch along that gradient direction (requires a couple of experiments that we omit in our Figures).

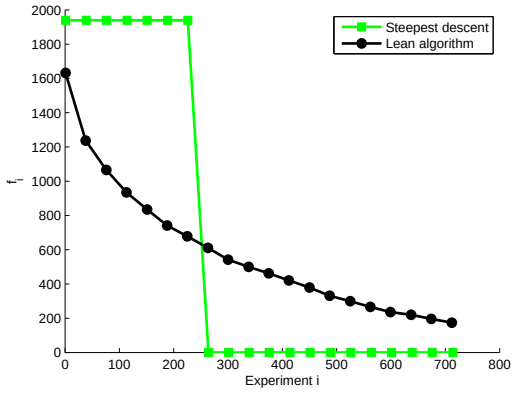


Figure 12: Lean algorithm using adaptive permutation versus steepest descent

In Figure 12, the x axis represents the successive experiments performed (step after step) and the y axis the correspondant function response for each experiment. *Since it would be messy to display one point for each experiment, the points are a local average of the experiments performed. For example, the first point represent the average of the* 33 *first experiments (equivalent to the* 3 *first steps), the second the* 33 *next ones, and so on. That is why, despite the fact we start from the same center point, the curves does not start from exactly the same value on the graphs.*

We see that it is not worth using the lean algorithm if one can afford doing more than 250 experiments, since then a simple gradient solver finds the exact solution in 1 single step (not taking into account the couple of experiments used for the linesearch). The lean algorithm can however be preferable in the case one can afford only 100 experiments; it will then be able to give a solution about 50% better than the starting point.

Of course, that example is a circular paraboloid that makes the gradient solver especially efficient. We now spice it a bit up in this new fashion:

$$\begin{aligned} \min_x f(x) &= \frac{1}{1000} \sum_{i=1}^{N} e^{\frac{1}{50} i} (x_{\sigma(i)} - i)^2 \qquad \text{(1 bis)} \\ -100 \leq x_i &\leq 250, \quad i = 1, ..., N \\ N &= 250 \end{aligned}$$

This function has now narrow valleys. Results are displayed in figure 13.

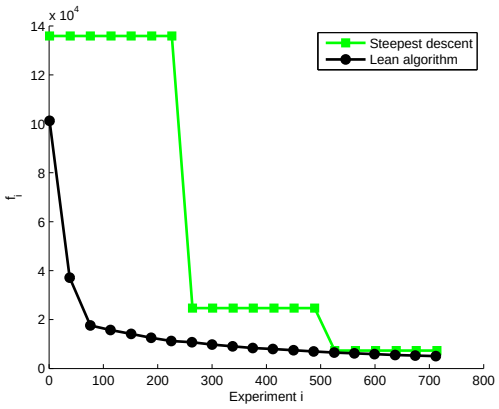
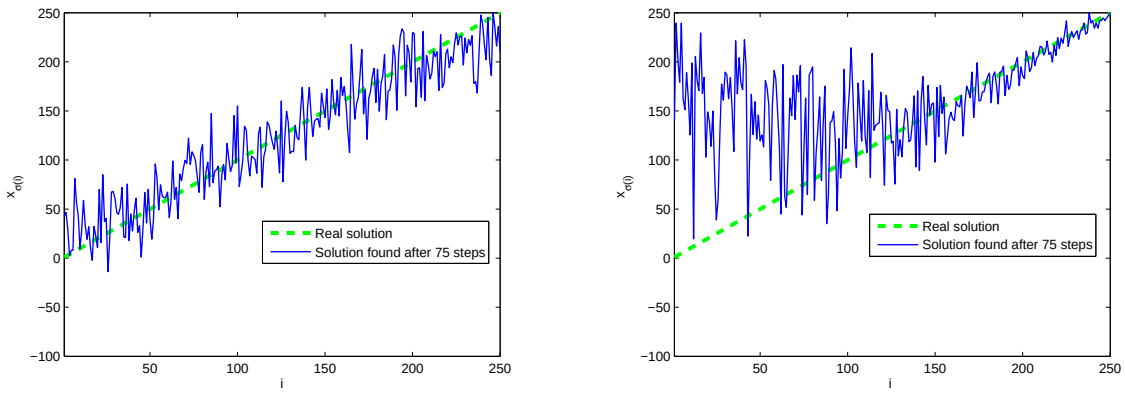


Figure 13: Lean algorithm using adaptive permutation versus steepest descent

Now the steepest descent is not converging that fast, and we see that the lean algorithm is actually converging faster, as long as the number of experiments is limited to around 600. The result is especially interesting if one can for example only afford to perform 50 experiments. Then the best solution found is 16680, which is still "far" from 0 the real optimum, but which can be compared with the value at the starting point: 135870. That represents an improvement of more than 81% while performing a number of experiments less than a fifth of the number of variables. To get some ideas on the behaviour of the algorithm, check Figure 14. In it are represented the solutions found by the algorithm after 75 steps for problem (1) and (1 bis).



(a) Solution for test problem 1

(b) Solution for test problem 1 bis

Figure 14: Solutions found after 75 steps

In Figure 14(a), we see that the error between the calculated solution and the real one is spread equally on all parameters. But in Figure 14(b), that error is very small for the parameters from $\sigma(200)$ to $\sigma(250)$ but gets larger for the ones from $\sigma(1)$ to $\sigma(200)$. Actually, we can roughly say that the error per parameter decreases with its importance. The importance of the parameters in problem 1 bis is indeed following the term $e^{\frac{1}{50}i}$. This means that the algorithm is taking care

of the parameters having the biggest influence at first. That implies that the algorithm will be particularly fast and efficient on problems where few parameters influence most of the system.

Test problem 2:

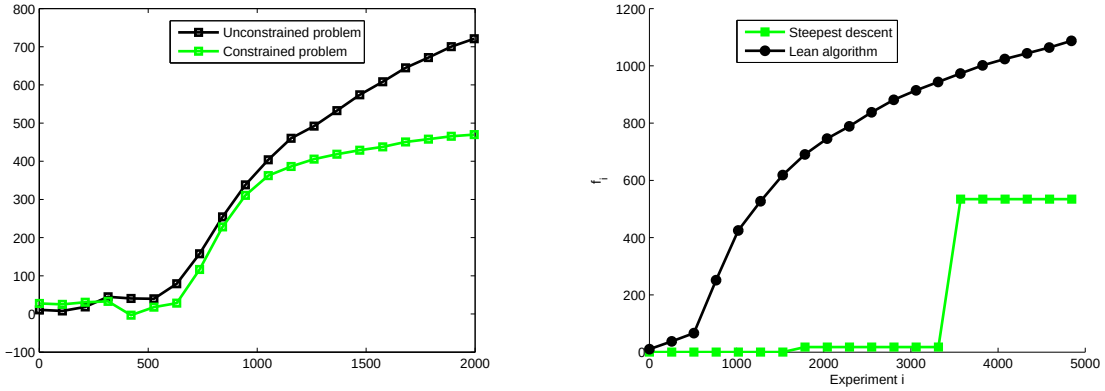
Consider now the unconstrained problem:

$$\max_x f(x) = \sum_{i=1}^{N} \sin(ix_i) \qquad \text{(2 UNC)}$$
$$-\frac{3\pi}{4} \leq x_i \leq \frac{3\pi}{4}, \quad i = 1, ..., N$$
$$N = 1700$$

and a constrained version:

$$\max_x f(x) = \sum_{i=1}^{N} \sin(ix_i) \qquad \text{(2 CONST)}$$
$$-\frac{3\pi}{4} \leq x_i \leq \frac{3\pi}{4}, \quad i = 1, ..., N$$
$$\sum_{i=1}^{1500} \sin(ix_i) \leq 300$$
$$N = 1700$$

Here, the objective function is a sum of sinus function, with increasing frequences. Small variations of each parameter has a different influence on the response. For example, changing x_{1700} from 0 to $\frac{1}{1700}\frac{\pi}{2}$ adds 1 to the response, but doing such a change on x_1 has almost no influence ($|\sin(\frac{1}{1700}\frac{\pi}{2})| \approx \frac{1}{1700}\frac{\pi}{2} \leq 10^{-3}$). For the unconstrained problem, the optimum is 1700 and it is reached when each sinus is 1 (there are hence several solutions). For the constrained problem, the optimum is then obviously only 500.
We use the same design than in the previous example and solve both problems. Results obtained after 300 steps and $300 \cdot 14 = 4200$ experiments are all displayed in Figure 15



(a) Constrained versus unconstrained problem

(b) Lean solver versus Gradient solver on the Unconstrained problem

Figure 15: Performance of the lean algorithm

In Figure 15(a), we can see that the constrained problem reaches its optimum in almost less

experiments than the number of variables in the problem. Once the algorithm gets close to an optimum, many unfeasible experiments actually occur, since the algorithm is stopped by the constraint on its way and is hence standing just against it.
In the unconstrained case, the algorithm continues its way, and much faster than the steepest descent as can be seen in Figure 15(b) (especially for the 3000 first experiments).

Test problem 3: pairwise interactions

Consider the problem:

$$\begin{aligned} \min_x f(x) &= \sum_{i=1}^{N-1} x_{\sigma(i)} x_{\sigma(i+1)} \\ -2 &\leq x_i \leq 5, \quad i = 1, ..., N \\ N &= 1700 \end{aligned}$$

where σ is some permutation of $\{1, ..., 250\}$, just introduced for not to give any possible advantage coming from some ordering to the algorithm (as used in the *Test problem* 1). This function is the sum of pairwise interactions of the vector x. It has multiple saddle points and its minimum is -16990. It is reached when the values of x are alternatively -2 and 5, with respect to σ. On such an example, the steepest descent algorithm has a very poor convergence.

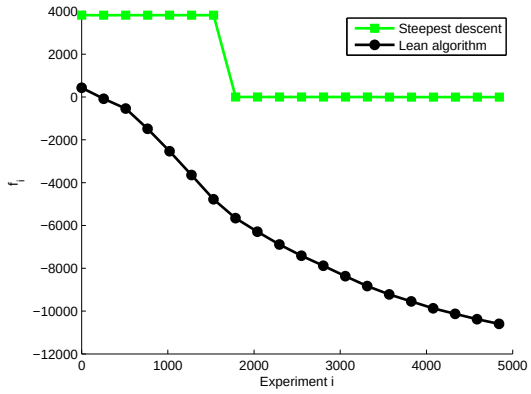
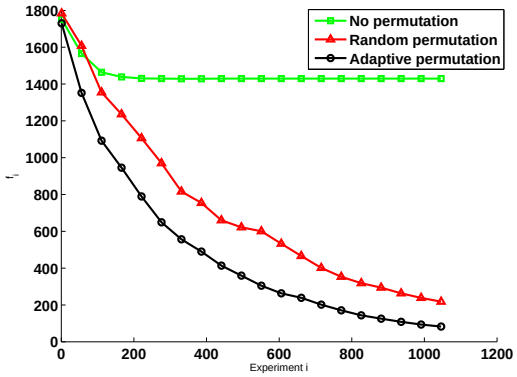


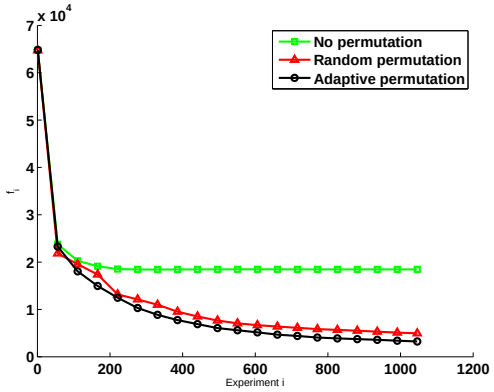
Figure 16: Convergence of the lean algorithm versus steepest descent

In Figure 16, we see that the steepest descent stops after 1 step at 0. The lean algorithm is doing much better than that, and again, even before the steepest descent has time to do one single step, the lean algorithm has already improved significantly the objective function.

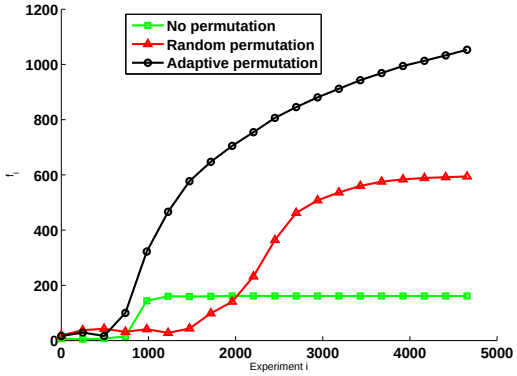
6.1.2 Overview of the performance of the lean algorithm with different design permutation strategies and different design sizes



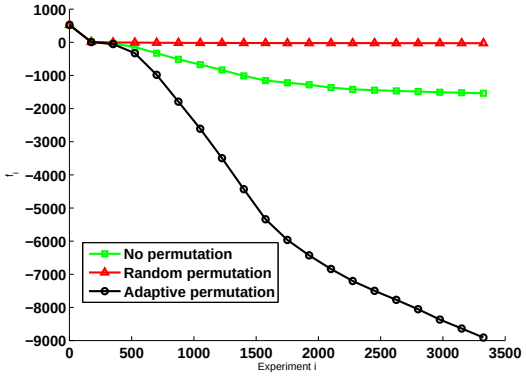
(a) Test problem 1: minimization
(Circular paraboloid)



(b) Test problem 1 bis: minimization
(Paraboloid with narrow valleys)



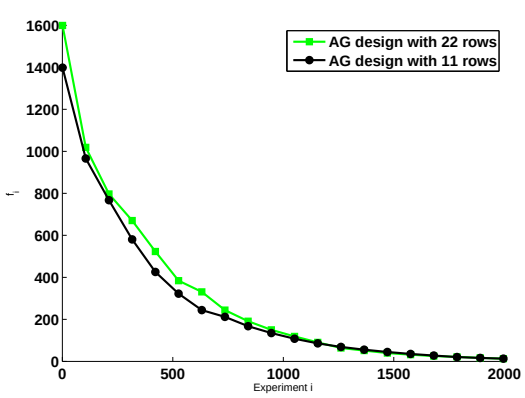
(c) Test problem 2: maximization
(Sum of sinus functions with increasing frequency)



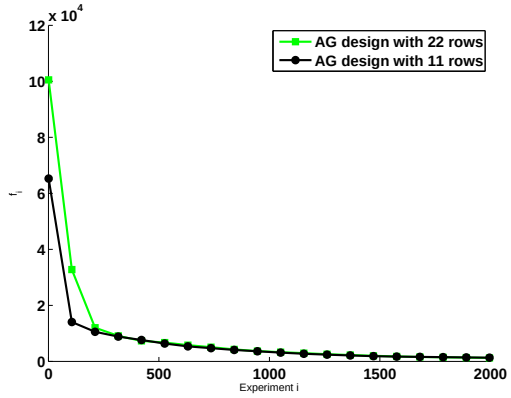
(d) Test problem 3: minimization
(quadratic model restricted to interaction terms)

Figure 17: Convergence of the lean algorithm using different design permutation strategies

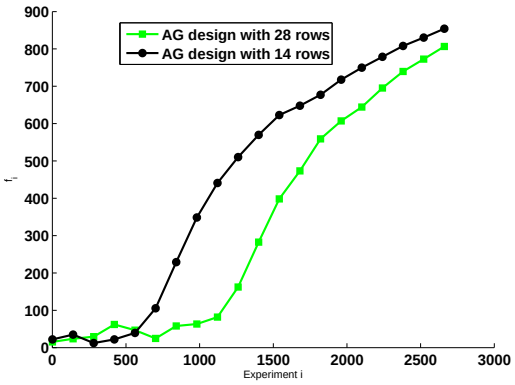
Figure 17 shows that without design permutation (that corresponds actually to applying the lean algorithm from [1] but with the AG design instead of the Lin design), the algorithm has a quick start but stops early after a couple of steps, and stays rather far from reaching the optimum.
With random permutation, the optimization continues along with the steps, and makes significant improvements from the no-permutation strategy in most cases.
As expected, the adaptive permutation seems the most efficient strategy, improving even more the speed of the random permutation.



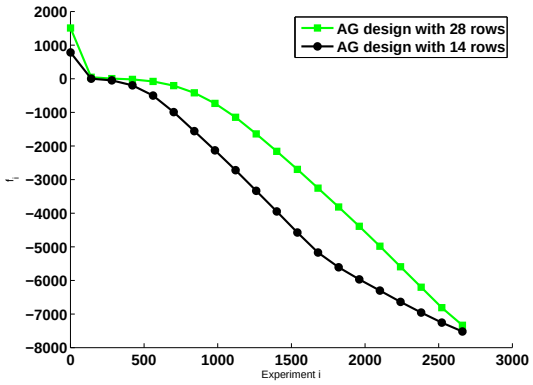
(a) Test problem 1: minimization
(Circular paraboloid)



(b) Test problem 1 bis: minimization
(Paraboloid with narrow valleys)



(c) Test problem 2: maximization
(Sum of sinus functions with increasing frequency)



(d) Test problem 3: minimization
(quadratic model restricted to interaction terms)

Figure 18: Convergence of the lean algorithm using adaptive permutation, but different design sizes

In Figure 18, we see that, as expected in section 5.2.3, taking the smallest design seems the best choice in almost all cases. Using a design twice bigger does not give twice more information but just $\sqrt{2}$ times more (from Ahlinder and Gustafsson [2]). That is not sufficient to compensate the fact that 2 times more experiments were performed.

6.2 On an industrial application

We will know try the algorithm on a concrete example issued from Volvo Technology. We will just give a small description of the problem, without going into details. ICES (Internal Combustion Engine Simulation Program) is a simulation program for four-stroke gasoline or diesel engines. Models of engine can be built from components like pipes, cylinders, valves, turbine, compressor, etc... Each of these components has several parameters that affect the efficiency of the whole engine. The program is able to generate the performance for the complete engine.

To create our optimization problem, we first picked some engine model, and then selected 162 relevant and possibly changeable parameters out of its components. We also decided on a reasonnable range of variation for each parameter. With speed and fuel consumption fixed to some value, keeping emissions, temperature and pressure below the starting value, our goal is to maximize the *Brake mean effective pressure*, which is the ratio between power and size of the engine. We hence simply try to make a more efficient engine without increasing emissions or decreasing lifetime.
The 162 parameters we have selected have preliminary values that have been pre-established by engineers and are used as a starting point.

Results are displayed in Figure 19.

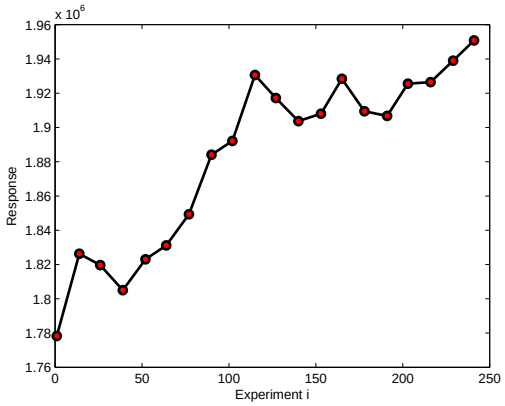


Figure 19: Convergence of the algorithm on the engine optimization problem

We see that the system is improved, but the improvement is not done regularly. It comes from the fact that the algorithm sometimes choose a direction that actually decrease the objective function to find back feasibility. Since they are several tough constraints to fulfill, that often happens.

Despite that, the improvement is noticeable: the value of the objective increases from about 1.78 to 1.93 MegaPascal (about 8% improvement) in just about 125 experiments which is less than the number of parameters in our problem (162).
The most significant parameters were the bore and the stroke. This means that the engine *brake mean effective pressure* was mainly increased by downsizing (making the engine smaller means increasing the ratio between power and engine's size). The total power was still slightly increased without affecting fuel consumption i.e. we got a slightly higher efficiency of the engine.

7 Conclusion

The lean optimization algorithm presented in [1] was an interesting first approach for optimizing very large problems with few experiments. In this paper, we have presented a new algorithm, keeping the same basical idea, but also giving some new hints for getting significant improvements.

We also presented the Ahlinder-Gustafsson supersaturated design (that can actually be called *fully saturated* since it is the most supersaturated one that still keeps relevancy). That design would be considered as bad according to the $E(s^2)$ criteria defined in [4], but its efficiency when used in the lean algorithm gives credit to [2].

The main conclusions on the new lean algorithm are:

- it should be used with **the smallest AG design possible**
- **the adaptive design permutation** improves almost always its performance.
  Note that adaptive permutation makes sense only if used with a column-ordered design, such as the AG design
- it is especially efficient for problems where the response is mainly affected by a little percentage of the parameters (for example 20%).
- it improves the performance of the previous version presented in [1], even when using the AG design instead of the Lin design [4]. Unlike the former method, it keeps converging after the first few steps (thanks to the new **shrinking strategy**), which is particularly beneficial when the number of parameters is beyond 1000.
- it converges much faster than a simple gradient solver as long the problem is nasty enough (for example, not a simple circular parabolic problem).

Acknowledgements

We would like to thank Ivar Gustafsson from the Department of Mathematical Sciences, Chalmers University of Technology, Göteborg, Sweden, for his valuable comments, and Volvo Technology Corporation, Göteborg, Sweden, for supporting this work.

References

[1] I.Siomina, S.Ahlinder, Lean Optimization using supersaturated design, Applied Numerical Mathematics 58 (1) (2008) 1-15

[2] S.Ahlinder, I.Gustafsson, On Super Saturated Experimental Design, (to be published)

[3] D.K.J. Lin, A new class of supersaturated designs Technometrics 35 (1993) 28-31

[4] D.K.J. Lin, Generating systematic supersaturated design, Technometrics 37 (2) (1995) 213-225

[5] M Liu, R Zhang, Construction of E(s^2) optimal supersaturated designs using cyclic BIBDs, Journal of Statistical Planning and Inference 91 (2000) 139-150

[6] MQ Liu, D.K.J. Lin, Construction of optimal mixed-level supersaturated designs, Statistica Sinica 19 (2009), 197-211

[7] D.Jones, M.Schonlau, W. Welch, Efficient global optimization of expensive black-box functions, Journal of Global Optimization 34 (3) (2006) 441-466

[8] E. H. Moore,On the reciprocal of the general algebraic matrix, Bulletin of the American Mathematical Society 26 (1920) 394-395

[9] Roger Penrose, A generalized inverse for matrices, Proceedings of the Cambridge Philosophical Society 51 (1955) 406-413